BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

Nachdem also das Terminal den für diese Karte notwendigen Authentisierungsschlüssel errechnet hat, folgt der übliche Ablauf im Rahmen des Challenge-Response-Verfahrens. Die Chipkarte erhält vom Terminal eine Zufallszahl, verschlüsselt diese mit ihrem kartenindividuellen Schlüssel und sendet sie an das Terminal zurück. Dieses führt analog der Chipkarte die Umkehrfunktion der Berechnung aus und vergleicht die beiden Ergebnisse. Stimmen sie überein, so besitzen sowohl Chipkarte als auch Terminal ein gemeinsames Geheimnis, nämlich den geheimen kartenindividuellen Schlüssel, und die Chipkarte ist vom Terminal authentisiert.

Der Vorgang der Authentisierung ist durch den Aufruf von DES-Algorithmen und die Datenübertragung von und zur Karte etwas zeitintensiv. Dies kann bei manchen Anwendungen zu Problemen führen.

Unter den folgenden Annahmen kann man den Zeitbedarf für eine einseitige Authentisierung überschlagsmäßig errechnen. Angenommen sei eine Chipkarte mit einem Takt von 3,5 MHz, dem Übertragungsprotokoll T=1, einem Teiler von 372 und einem DES-Algorithmus, der 17 ms für einen Block benötigt. Alle internen Routinen der Chipkarte seien hier ohne genauere Angabe mit 9 ms angenommen, was die Berechnung vereinfacht, das Endergebnis aber nur unwesentlich verfälscht.

Tabelle 4.17 Berechnung des Zeitbedarfs in der Chipkarte für eine einseitige Authentisierung unter Berücksichtigung der Übertragungszeit.

Kommando	Zeitbedarf für Übertragung		Zeitbedarf für Berechnung	
INTERNAL AUTHENTICATE	38,75 ms	+	26 ms	= 64,75 ms

Man sieht deutlich anhand der Berechnung, daß eine einzige Authentisierung ca. 65 ms benötigt. Dies kann im Normalfall in einer Anwendung ohne zeitliche Probleme ausgeführt werden.

4.10.2 Gegenseitige symmetrische Authentisierung

Das Prinzip der gegenseitigen Authentisierung (*mutual authentication*) beruht auf einer zweifachen einseitigen Authentisierung. Man könnte auch zwei einseitige Authentisierungen abwechselnd für beide Kommunikationspartner ausführen. Dies wäre dann im Prinzip eine gegenseitige Authentisierung. Da jedoch der Kommunikationsaufwand aus zeitlichen Gründen so gering wie möglich gehalten werden muß, gibt es ein Verfahren, in dem die beiden einseitigen Authentisierungen miteinander verflochten sind. Dabei erzielt man auch noch eine höhere Sicherheit als mit zwei nacheinander ausgeführten Authentisierungen, da es für einen Angreifer viel schwieriger ist, in den Kommunikationsablauf einzugreifen.

Damit das Terminal mit dem Hauptschlüssel aus der Kartennummer den kartenindividuellen Authentisierungsschlüssel berechnen kann, benötigt es als erstes die Kartennummer. Nachdem das Terminal die Kartennummer erhalten hat, berechnet es den individuellen Authentisierungsschlüssel für diese Chipkarte. Dann fordert es von der Chipkarte eine Zufallszahl an und generiert selbst ebenfalls eine Zufallszahl. Nun setzt das Terminal beide Zufallszahlen vertauscht hintereinander, verschlüsselt sie mit

dem geheimen Authentisierungsschlüssel und sendet den erhaltenen Schlüsseltext zur Karte. Das Vertauschen hat den Zweck, Challenge und Response unterschiedlich zu machen.

Diese kann den erhaltenen Block entschlüsseln und prüfen, ob die vorher an das Terminal gesendete Zufallszahl mit der zurückerhaltenen übereinstimmt. Ist dies der Fall, so weiß die Chipkarte, daß das Terminal den geheimen Schlüssel besitzt. Damit ist das Terminal gegenüber der Chipkarte authentisiert. Daraufhin vertauscht die Chipkarte die beiden Zufallszahlen, verschlüsselt sie mit dem geheimen Schlüssel und schickt das Ergebnis zum Terminal.

Das Terminal entschlüsselt den erhaltenen Block und vergleicht die zuvor an die Chipkarte gesendete Zufallszahl mit der erhaltenen. Stimmt diese mit der vormals gesendeten überein, so ist auch die Chipkarte gegenüber dem Terminal authentisiert. Damit ist die gegenseitige Authentisierung abgeschlossen, und sowohl Chipkarte als auch Terminal wissen, daß der jeweilig andere vertrauenswürdig ist.

Um den Zeitbedarf der Kommunikation zu minimieren, kann die Chipkarte zusätzlich zur Kartennummer auch noch die Zufallszahl zurücksenden. Dies ist dann von Interesse, wenn die gegenseitige Authentisierung zwischen Chipkarte und einem Hintergrundsystem stattfindet. Die Chipkarte wird dabei direkt vom Hintergrundsystem transparent zum Terminal angesprochen. Die Datenübertragungsgeschwindigkeit ist dabei oft sehr niedrig, und der Kommunikationsablauf muß dadurch so stark wie möglich vereinfacht werden.

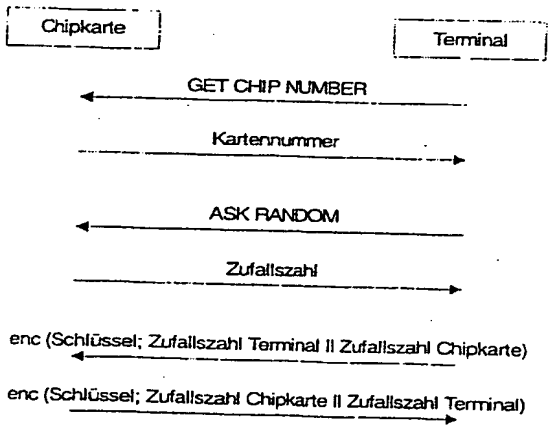

Bild 4.47 Die gegenseitige Authentisierung mit kartenindividuellem Schlüssel und einem symmetrischen Kryptoalgorithmus. Der Ablauf entspricht einer gegenseitigen Authentisierung von Chipkarte und Terminal, wie sie durch das Kommando MUTUAL AUTHENTICATE nach ISO/IEC 7816-8 realisiert werden kann.

Um den erheblichen Zeitaufwand, auch im Gegensatz zur einseitigen Authentisierung, aufzuzeigen, ist im folgenden nochmals eine rechnerische Betrachtung aufgeführt. Die zugrundeliegenden Annahmen sind dabei analog der einseitigen Authenti-

sierung. Man sieht, daß die gegenseitige Authentisierung beinahe dreimal so lange dauert als eine einseitige.

Tabelle 4.18 Berechnung des Zeitbedarfs in der Chipkarte für eine gegenseitige Authentisierung unter Berücksichtigung der Übertragungszeit. Es wurde angenommen, daß keine abgeleiteten Schlüssel Verwendung finden (GET CHIP NUMBER ist deshalb nicht notwendig).

Kommando	Zeitbedarf für Übertragung		Zeitbedarf für Berechnung	
ASK RANDOM	28.75 ms		26 ms	
MUTUAL AUTHENTICATE	68.75 ms		95 ms	
	97,50 ms	+	121 ms	= 218,50 ms

4.10.3 Statische asymmetrische Authentisierung

Nur sehr wenige Chipkarten-Mikrocontroller besitzen eine Recheneinheit, mit der RSA-Berechnungen durchgeführt werden können. Dies liegt vor allem daran, daß diese zusätzlichen Platz auf dem Chip benötigt, was den Preis erhöht.

Da nun aber ein zusätzliches asymmetrisches Authentisierungsverfahren vermehrten Schutz bedeutet, da ein Angreifer nicht nur einen kryptografischen Algorithmus brechen muß, sondern zwei, möchte man oft noch diese Art von Authentisierung verwenden. Um das Problem der nicht vorhandenen Recheneinheit auf der Chipkarte zu umgehen, fand man als Ausweg eine statische Authentisierung der Chipkarte durch das Terminal. Diese erfordert lediglich eine Verifizierung im Terminal. Eine zusätzliche Recheneinheit auf dem Terminal erhöht aber die Kosten im Verhältnis zum Gesamtpreis dieser Geräte nur unwesentlich, deshalb ist dieser Weg wesentlich kostengünstiger als die Verwendung spezieller Chipkarten-Mikrocontroller. Zudem ist das Verfahren wesentlich schneller, da nur eine asymmetrische Verschlüsselung notwendig ist, im Gegensatz zu zwei bei einer dynamischen asymmetrischen Authentisierung.

Man erkauft sich diesen Kompromiß jedoch durch eine verminderte Sicherheit des Authentisierungsverfahrens. Ein Schutz gegen Wiedereinspielung ist durch das statische Verfahren natürlich nicht gegeben. Deshalb benutzt man es auch nur als zusätzliche Überprüfung der Authentizität der Karte, die vorher schon mit einem dynamischen symmetrischen Verfahren überprüft worden ist.

Das Verfahren funktioniert in seinem grundlegenden Ablauf folgendermaßen: Bei der Personalisierung werden in jede Chipkarte kartenindividuelle Daten eingetragen. Dies sind beispielsweise eine Kartennummer, der Name des Kartenbesitzers und seine Adresse. Über diese Daten, die während der Lebensdauer der Karte nicht veränderbar sind, wird während der Personalisierung eine digitale Signatur mit einem geheimen Schlüssel gerechnet. Der Schlüssel wird im System global verwendet. Benutzt man nun diese Karte an einem Terminal, so liest dieses aus einer Datei auf der Karte die Signatur und die signierten Daten aus. Das Terminal besitzt den für alle Chipkarten gültigen öffentlichen Schlüssel und kann die gelesene Signatur verschlüsseln und das Ergebnis

5.10 Chipkarten-Betriebssysteme mit nachladbarem Programmcode

Der Abschnitt „nachladbarer Programmcode" umfaßte 1995 in der ersten deutschen Auflage dieses Buches genau 642 Wörter, geschrieben in 65 Zeilen. Der Textumfang ist in dieser Auflage mittlerweile auf das 18fache gestiegen. Allein dieser Punkt zeigt, wie wichtig dieses Thema mittlerweile geworden ist. Man kann wohl ohne zu übertreiben behaupten, daß sich hier innerhalb eines Jahres (1997/1998) ein Paradigmenwechsel vollzogen hat. Nachladbarer Programmcode in Chipkarten wird mittlerweile als Regelfall und nicht mehr als Ausnahme angesehen, obwohl er noch von den wenigsten Chipkarten-Betriebssystemen unterstützt wird.

Die Gründe, warum das Nachladen von ausführbarem Programmcode so stark an Bedeutung gewonnen hat, sind auch rückblickend nicht ganz eindeutig nachvollziehbar. Eine Triebfeder mag der im Jahr 1994 bekanntgewordene Rechenfehler (FDIV-Befehl) in den damals weit verbreiteten Pentium-Prozessoren gewesen sein. Eine Ausbesserung per Software-Download war nicht möglich, da es ein echter Fehler in der Hardware war. Allerdings gab es für viele Anwendungsprogramme Patches, um den Fehler zu umgehen.

Es ist wahrscheinlich, daß dieser Fehler dazu führte, daß mit geringer zeitlicher Verzögerung einige große Systembetreiber plötzlich die Möglichkeit vorsahen, ausführbaren Programmcode in Chipkarten nachzuladen. Eine der größten Anwendungen mit ausführbarem Programmcode ist die ec-Karte mit Chip in Deutschland. Allerdings wird diese technische Möglichkeit zur Zeit nicht genutzt und stellt De-facto nur einen Rettungsanker für eventuell auftretende schwerwiegende Programmfehler dar. Auch bei GSM existieren Betriebssysteme, die es ermöglichen, daß Programmcode für spezielle Anwendungen über die Luftschnittstelle nachgeladen werden kann.

Im Gegensatz zu allen anderen Betriebssystemen für Computer ist es aber nicht generell üblich, Programme in Chipkarten nach Ausgabe einzubringen und dort bei Bedarf auszuführen. Dies ist aber eigentlich neben der Datenspeicherung eine der Hauptfunktionen aller Betriebssysteme. Es gibt natürlich Gründe, warum bisher gerade diese Funktionalität in der Chipkartenwelt weitgehend gefehlt hat.

Technisch und funktionell gesehen stellt ausführbarer Programmcode, beispielsweise in Dateien (d.h. EFs) gespeichert, keinerlei Problem dar. Neuere Betriebssysteme bieten deshalb auch die Möglichkeit, Dateien mit ausführbarem Code zu verwalten und auch zu einem Zeitpunkt nach der Personalisierung in die Chipkarte zu laden. Damit ist es möglich, daß beispielsweise ein Anwendungsanbieter Programmcode in der Chipkarte ausführen kann, den der Betriebssystem-Hersteller nicht kennt. So kann ein Anwendungsanbieter einen nur ihm selbst bekannten Verschlüsselungsalgorithmus in die Chipkarte einbringen und dort ausführen. Hierdurch ist es möglich, das Wissen um die Sicherheitsfunktionen des Systems auf verschiedene Parteien zu verteilen, was eine Grundforderung in Sicherheitssystemen ist.

Ein weiterer gewichtiger Grund für den Mechanismus des nachladbaren Programmcodes ist die sich damit eröffnende Möglichkeit der Beseitigung von Programm-

fehlern (*bug-fixing*) in vollständig personalisierten Karten. Erkannte Fehler im Betriebssystem können damit bei bereits ausgegebenen Karten behoben oder zumindest entschärft werden.

Es gibt grundsätzlich zwei Wege, Programmcode in einer Chipkarte auszuführen. Der erste und technisch einfachste Weg ist, compilierten Code in der Maschinensprache des Zielprozessors (*native code*) in Dateien der Chipkarte zu laden. Dieser Programmcode muß natürlich relokierbar sein, da die Speicheradressen nach außen nicht bekannt sind. Neben der technischen Unkompliziertheit dieser Lösung kann der Programmcode noch mit voller Ausführungsgeschwindigkeit des Prozessors abgearbeitet werden, was diese Lösung gerade für nachladbare Algorithmen sehr interessant macht. Weiterhin ist auch kein zusätzlicher Programmcode für einen Interpreter in der Chipkarte notwendig. Das große Problem dieser Lösung ist, daß der nachgeladene Programmcode bei Mikrocontrollern ohne MMU (*memory management unit*) auch auf Speicherbereiche von Fremdanwendungen zugreifen kann.

Der zweite Weg, ausführbaren Programmcode in Chipkarten auszuführen, besteht darin, ihn zu interpretieren. Der Interpreter prüft dann während der Programmausführung, welche Speicherbereiche angesprochen werden. Die Interpretation muß aber schnell ablaufen, da ein langsam ausgeführter Programmcode keine Vorteile mehr bringt. Ebenso soll die Implementation eines Interpreters selber sowenig Speicher wie möglich in Anspruch nehmen, da dieser bekanntermaßen stark limitiert ist. Die derzeitig bekanntesten Lösungsvarianten dazu sind die Java Card Spezifikation [Javasoft, JCF] und der C-Interpreter MEL (*Multos executable language*) von Multos [Maosco]. Mittlerweile gibt es für Chipkarten sogar einen BASIC-Interpreter [Zeitcontrol]. Interpreter, die dem Programm einer Anwendung einen eigenen geschützten Speicher zur Verfügung stellen, sind im übrigen nicht für Fehlerbeseitigungen im Betriebssystem von Chipkarten geeignet, da sie auf diese Programm- und Datenteile konsequenterweise keinen Zugriff haben.

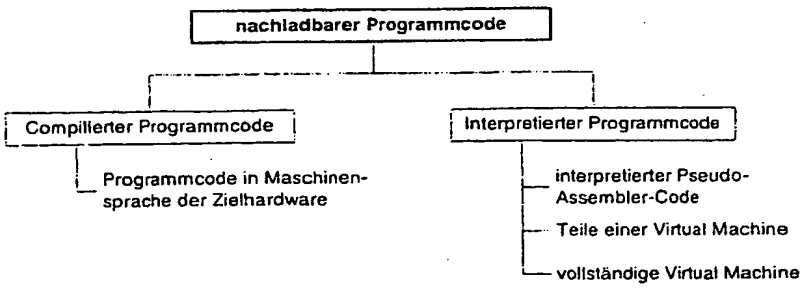

Bild 5.27 Klassifizierungsbaum der Varianten, um ausführbaren Programmcode in Chipkarten-Betriebssysteme nachzuladen und dort auszuführen.

Das Urproblem aller Interpreter ist jedoch die langsame Abarbeitungsgeschwindigkeit, welche für dieses Prinzip immanent ist. Um dieses Minus auszugleichen und um den Programmcode für den eigentlichen Interpreter so klein wie möglich zu halten,

gibt es mehrere Lösungsansätze. Die einfachste Methode ist es, einen Pseudocode zu interpretieren, welcher idealerweise den Maschinenbefehlen der Zielhardware möglichst ähnlich sein sollte. Die Abarbeitungsgeschwindigkeit ist durch die maschinennahe Pseudosprache verhältnismäßig hoch, und es kann maschinenunabhängiger Programmcode benutzt werden. Speicherzugriffe während der Interpretation können überwacht werden, was aber nicht zwangsläufig der Fall sein muß. Eine langsamere und programmiertechnisch etwas aufwendigere Lösung ist die Aufspaltung des Interpreters in einen offcard-Teil (*offcard virtual machine*) und einen oncard-Teil (*oncard virtual machine*). Dieser Lösungsweg wird bei vielen heutigen Java Card Implementationen beschritten. Er hat den großen Vorteil eines verläßlichen Speicherschutzes und vollständiger Hardwareunabhängigkeit. Nachteilig ist die Aufteilung des Interpreters in off- und oncard-Teil. Dies bedingt zwangsläufig einen kryptografischen Schutz beim Übertragen von Programmen zwischen offcard-Teil und oncard-Teil des Interpreters, da mit manipuliertem Programmcode der oncard-Teil des Interpreters bewußt zu einem Fehlverhalten gebracht werden kann.

Die technisch optimale Lösung ist ein vollständiger Interpreter auf der Chipkarte. Damit ist es möglich, in die Chipkarte beliebigen Programmcode zu laden und dort ohne Risiko für andere auf der Chipkarte befindliche Anwendungen auszuführen. Allerdings ist der Programmumfang dazu auch entsprechend groß, weshalb es sicherlich noch einige Jahre und mehrere Chipgenerationen dauern wird, bis diese Variante breiten Einzug in die Chipkartenwelt hält.

5.10.1 Executable Native-Code

Mikrocontroller für Chipkarten haben zur Zeit meistens Prozessoren, die über keinerlei Speicherschutzmechanismen oder Überwachungsmöglichkeiten verfügen. Sobald sich der Programmzähler innerhalb eines fremden Maschinencodes für den Prozessor befindet, liegt die gesamte Kontrolle aller Speicher und Funktionen bei diesem ausführbaren Code. Es gibt dann keinerlei Möglichkeiten mehr, dieses ausführbare Programm in seinen Funktionen zu beschränken. Jede adressierbare Speicheradresse kann unter Umgehung aller Speichermanager oder Handler gelesen und – sofern im RAM oder EEPROM – auch geschrieben werden. Alle Speicherinhalte können dann natürlich auch über die Schnittstelle zum Terminal gesendet werden.

Genau dies ist die Schwachstelle bei ausführbaren und nachladbaren Programmen. Würde man jedermann ein Nachladen von Programmen erlauben, oder wäre es durch Umgehung der Schutzmechanismen möglich, dann ist keine Sicherheit mehr für geheime Schlüssel oder Informationen innerhalb des gesamten Speicherbereichs gegeben. Dies wäre der ideale Angriff auf eine Chipkarte. Diese Karte würde sich nach außen hin wie eine nicht manipulierte Karte verhalten, und mit einem speziellen Kommando könnten der gesamte Speicher ausgelesen oder Teile davon geschrieben werden.

Wäre das Laden nur einigen wenigen Anwendungsanbietern erlaubt – was mit einer gegenseitigen Authentisierung vor dem eigentlichen Laden des Programmcodes ohne weiteres durchführbar ist – so ist das Problem auch nicht aus der Welt geschafft. Der Anwendungsanbieter kann ohne Einschränkungen über die Grenzen seines ihm zu-

geteilten DFs auf geheime Informationen von anderen vorhandenen Anwendungen zugreifen. Das System wäre wiederum gebrochen.

Doch gibt es noch ein weiteres stichhaltiges Argument gegen von Dritten nachladbaren Programmcode. Um wichtige Funktionen im Betriebssystem nutzen zu können, muß der Ersteller der nachladbaren Datei alle Einsprungadressen und Aufrufparameter kennen. Die Betriebssystemhersteller erachten es aber für die Sicherheit als relevant, daß sowenig wie möglich über interne Abläufe oder Adressen von Programmcode bekannt ist. Zusätzlich müßte auch noch sichergestellt werden, daß der eingebrachte Code genau das fehlerfrei ausführt, was beabsichtigt ist, und nicht etwa ein trojanisches Pferd enthält. Dies kann dann wiederum nur eine unabhängige Instanz prüfen.

Die eleganteste und auch die wohl zukunftsträchtigste Lösung ist der Einsatz einer hardware-unterstützten Speicherverwaltung (*memory management unit – MMU*) zusätzlich zum eigentlichen Prozessor in der Chipkarte. Diese prüft den ablaufenden Programmcode mit einer Hardwareschaltung auf die Einhaltung seiner ihm zugewiesenen Grenzen. Erst dann wäre es ohne den Verlust an Sicherheit möglich, jeden Anwendungsbetreiber Programmcode ohne vorherige Prüfung durch den Kartenherausgeber in die Chipkarte laden zu lassen. Diesem Anwender würde man den physikalisch zusammenhängenden Speicherbereich eines DFs zuweisen. Die MMU prüft die zugeordneten Speichergrenzen während des Aufrufs eines im DF nachgeladenen Programms. Werden die Grenzen überschritten, so kann man über einen Interrupt den Programmablauf unmittelbar stoppen und die Anwendung bis auf weiteres sperren.[1]

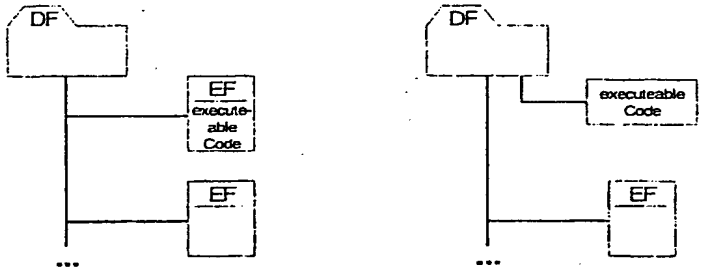

Bild 5.28 Die beiden unterschiedlichen Varianten zur Einbringung von ausführbarem Programmcode in ein übliches Chipkarten-Betriebssystem. Links als ausführbare Datei und rechts als ASC (*application specific commands*).

Für das Nachladen von nativem Programmcode in Chipkarten existieren zwei Varianten der Realisierung. In der ersten befindet sich der Programmcode in einem EF mit der Struktur „executable". Nach einer vorherigen Selektion ist das EF mit einem Kommando EXECUTE ausführbar. Je nach Anwendung ist dazu vorher noch eine Authentisierung notwendig. Die Parameter für den Programmaufruf sind im Kommando EXECUTE an die Chipkarte enthalten. Die vom Programm im EF erzeugte Antwort kommt als Teil der Antwort auf das Kommando zum Terminal zurück.

[1] siehe auch Abschnitt 3.4.3 Zusatzhardware

Die zweite Variante gestaltet sich vom Prinzip her etwas anders. Man benutzt dabei einen objektorientierten Ansatz. Dieser ist unter anderem auch in der EN 726-3 als *Application Specific Commands* (ASC) beschrieben. Nach dieser Norm enthält ein DF die vollständige Anwendung mit allen ihren Dateien und anwendungsspezifischen Kommandos. In einem in diesem DF intern vom Betriebssystem verwalteten Speicherbereich kann Programmcode nachgeladen werden. Dies geschieht mit einem speziellen Kommando, das alle dafür notwendigen Informationen an die Chipkarte sendet. Ist nun das betreffende DF selektiert und wird ein Kommando an die Karte gesendet, so prüft das Betriebssystem, ob es zu den Nachgeladenen gehört, und ruft gegebenenfalls ohne weiteres Zutun den im DF befindlichen Programmcode auf. Ist hingegen ein anderes DF selektiert, so existiert das nachgeladene Kommando in diesem Kontext nicht.

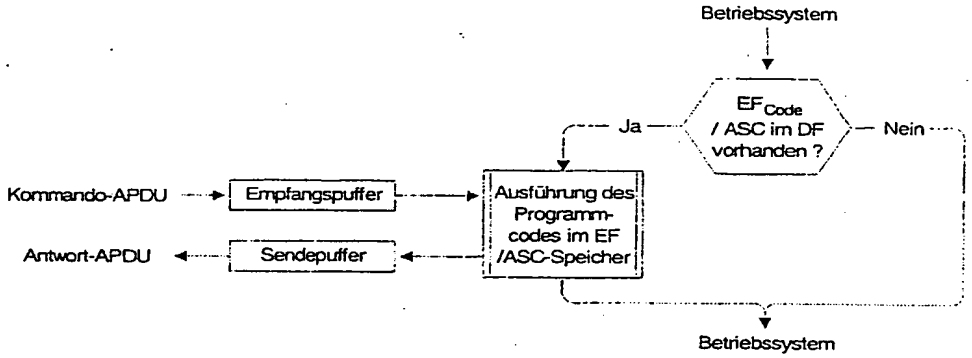

Bild 5.29 Die Grundlage des Aufrufverfahrens für in einem EF gespeicherten ausführbaren Programmcode bzw. Programmen im Rahmen von ASCs (*application specific commands*) in einem üblichen Multiapplication-Betriebssystem für Chipkarten.

Beispiel für in ein EF nachladbaren nativen Programmcode

Es existieren mehrere große Chipkarten-Anwendungen, deren spezifische Betriebssysteme ein Laden von ausführbarem Programmcode nach der Personalisierung vorsehen. Die Spezifikationen dazu sind aber in beinahe allen Fällen vertraulich, bzw. zum Teil ist bereits die Tatsache vertraulich, daß Programmcode nachgeladen werden kann. Deshalb sind in diesem Abschnitt die allgemeingültigen Grundlagen unabhängig von einem Betriebssystem aufgeführt und dann nachfolgend eine mögliche Realisierung im Detail dargestellt.

Der nachzuladende Programmcode muß einige Grundvoraussetzungen erfüllen, damit er überhaupt auf der Chipkarte ausgeführt werden kann. So trivial es klingen mag, aber die wichtigste Voraussetzung ist, daß der Prozessortyp (z.B.: 8051, 6805) bekannt ist. Gerade in einer heterogenen Umgebung mit vielen unterschiedlichen Chipkarten-Mikrocontrollern ist die Einhaltung dieser Forderung oft durchaus mit einigem Aufwand verbunden. Einher geht, daß ebenfalls das Betriebssystem der Chipkarte einschließlich API (*application programming interface*) mit allen Einsprungadressen, Über- und Rückgabeparametern bekannt sein muß.

Der nachzuladende Programmcode – es handelt sich dabei immer um Maschinencode der Zielprozessors (d.h. Native-Code) – muß entweder so programmiert sein, daß er relokatibel ist, oder die Chipkarte muß ihn während des Ladens on-the-fly selber relokieren. Die Forderung nach Relokatierbarkeit (d.h. Verschiebbarkeit des Programmcodes im Speicher) muß deshalb aufgestellt werden, weil die Speicheradresse für die Programmablage nur das Chipkarten-Betriebssystem kennt und der äußeren Welt unbekannt ist. In der Regel wird der Programmcode bereits bei der Softwareentwicklung so erstellt, daß er relokierbar ist. Dies bedeutet konkret, daß beispielsweise in diesem Programm keine Sprünge an absolute physikalische Adressen gemacht werden dürfen, sondern lediglich Sprünge relativ zur Adresse des Sprungbefehls.

Erfüllt der Programmcode alle diese Voraussetzungen, dann kann er prinzipiell in den Speicher einer Chipkarte geladen und dort ausgeführt werden. Der Programmcode kann natürlich je nach Bedarf aufgebaut sein. Eine mögliche Struktur zeigt das folgende Bild, wobei diese aber je nach Betriebssystem völlig verschieden sein kann. Das erste Datenelement in dem Beispiel ist ein eindeutiges Kennzeichen für das Chipkarten-Betriebssystem, daß es sich um Programmcode handelt. Allgemein bekannt ist so ein Kennzeichen auch als „Magic Number". Diese kurze Bytesequenz setzt sich beispielsweise bei Java-Class-Dateien aus den vier Bytes 'CAFEBABE' zusammen.

Im Anschluß an das Kennzeichen folgt bereits der Programmcode, welcher in diesem Beispiel noch mal in vier Teile aufgegliedert ist. Der erste Teil ist für alle notwendigen Initialisierungen, Sicherung von Daten und ähnliches vorgesehen. Nach dieser Startup-Routine folgt die eigentliche Funktionsroutine, welche den Programmcode für die gewünschte Aufgabe enthält. Daran schließt sich das Pendant zur Startup-Routine an, die Shutdown-Routine. Diese stellt sicher, daß das Programm korrekt abgeschlossen wird und führt dazu nach Bedarf eine Rücksicherung von Daten oder ggf. Änderungen auf dem Stack durch.

Optional und am Schluß dieser drei Programmteile befindet sich das vierte Programmelement. Es kann Programmcode enthalten, der resistent in die Software der Chipkarte eingebunden werden soll. Typischerweise würden hier Bugfixes für das Chipkarten-Betriebssystem abgelegt. Die vorangehenden drei Routinen würden Zeiger oder Handles so verändern, daß diese Programmroutinen permanent in die Programmabläufe des Betriebssystems eingebunden sind. Das Ganze verhält sich sehr ähnlich wie die noch aus DOS-Zeiten bekannten TSR-Programme (*terminate and stay resistant*), welche sich nach einem einmaligen Aufruf bis zum nächsten Reset fest im Betriebssystem verankerten. Die resistenten Programme in diesem Fall wären jedoch nach einmaligem Aufruf auf Dauer fest installiert und nicht nur für eine Sitzung.

Es wird hier davon ausgegangen, daß das nachgeladene Programm mit einem CALL-Befehl aufgerufen wird und mit einem RETURN-Befehl zum Aufrufer zurückkehrt. Prinzipiell wäre auch ein direkter Sprung mit JUMP auf den ersten Maschinenbefehl möglich, dies hätte aber den Nachteil, daß dann dem aufgerufenen Programm nicht bekannt ist, von wem es aufgerufen wurde.

Zur Absicherung gegen unbeabsichtigte Veränderung sollte der gesamte Datenblock noch mit einem Fehlererkennungscode (*error detection code – EDC*) abgesichert sein.

Alternativ dazu würde sich hier natürlich auch eine digitale Signatur als zusätzlicher Schutz anbieten. Die Chipkarte würde dann den öffentlichen Schlüssel besitzen und der Ersteller des Programmcodes den dazugehörigen geheimen Schlüssel. Damit wäre bindend sichergestellt, daß authentischer Programmcode auf der Chipkarte gestartet werden kann.

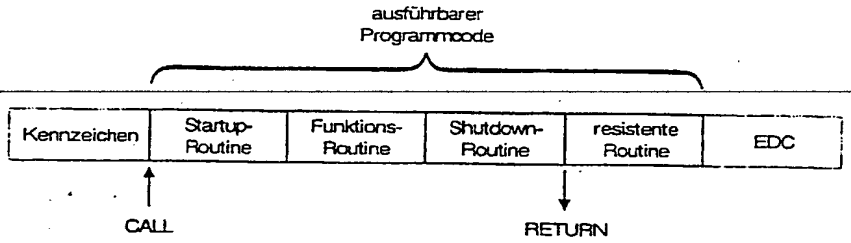

Bild 5.30 Möglicher Aufbau eines nativen und in ein EF nachladbaren und ausführbaren Programmcodes.

Der nachgeladene Programmcode kann entweder in einem EF gespeichert werden oder in einem für die äußere Welt unsichtbaren Speicherbereich für Programme innerhalb eines DFs. Im folgenden wird die erste Variante genauer aufgezeigt, da diese in der Praxis wesentlich häufiger anzutreffen ist.

Ideal zur Ablage von Programmcode sind EFs mit der Struktur „transparent" geeignet, da diese zweckmäßig mit dem UPDATE BINARY-Kommando und Offsetangabe in mehreren Teilstücken schreiben lassen. Zudem ist ihre maximale Größe von über 65 kByte selbst für umfangreiche Programme mehr als ausreichend. Diese EFs können die Eigenschaft „executable" haben, so daß in ihnen gespeicherter Programmcode direkt mit dem Kommando EXECUTE aufgerufen wird.

Manche Betriebssysteme besitzen aber auch eine von „transparent" abgeleitete Dateistruktur, welche sich dann „execute" nennt. Für die äußere Welt ist dies nicht von besonderer Bedeutung, zumal in der Regel auf beide Varianten mit UPDATE BINARY und EXECUTE zugegriffen werden kann. Mit dem FID bzw. Short-FID kann das EF selektiert werden. Die Zugriffsbedingung zum Lesen ist grundsätzlich auf „never" gesetzt. Das Schreiben von Daten ist in der Regel nur nach vorheriger Authentisierung und mit Secure Messaging erlaubt.

Die beiden Abläufe Bild 5.31 und Bild 5.32 zeigen im Überblick, wie Programmcode auf sichere Weise in das EF einer Chipkarte geladen werden kann. Sollte kein entsprechendes EF vorhanden sein, dann muß dieses vorher erzeugt werden. Im zweiten Ablaufbild ist überblickhaft dargestellt, daß das EF zuerst muß und anschließend der Programmcode mit dem Kommando EXECUTE gestartet werden muß. Das Kommando sieht optional eine Datenübergabe im Kommando-Body vor. Analog verhält es sich mit der Antwort, in welcher Daten bei Bedarf an das Terminal zurückgegeben werden können. Das aufgerufene Programm muß dazu natürlich die Daten aus dem Empfangspuffer lesen, auswerten und dann die Antwort in den Sendepuffer zurückschreiben.

Hintergrundsystem		Chipkarte
selektiere ein EF mit der Struktur execute	→	
gegenseitige Authentisierung von Hintergrundsystem und Chipkarte	↔	
schalte Secure Messaging ein		
sende n · UPDATE BINARY-Kommando mit dem Datenteil als ausführbaren Programmcode, geschützt mit Secure Messaging	→	

Bild 5.31 Ein möglicher Ablauf beim Laden von ausführbarem Programmcode in ein bereits vorhandenes EF mit der Struktur „execute". Die Zugriffsbedingung für UPDATE BINARY ist eine gegenseitige Authentisierung zwischen Chipkarte und Terminal sowie die Datenübertragung mit Secure Messaging.

Hintergrundsystem		Chipkarte
Selektiere EF mit ausführbarem Programmcode	→	
Sende EXECUTE-Kommando	→	– Prüfe Kennzeichen (*magic number*) des Programms – Prüfe EDC des Programms – Rufe den ersten Maschinenbefehl mit CALL auf

Bild 5.32 Ein möglicher Ablauf beim Starten von ausführbarem Programmcode. Dieser ist in diesem Beispiel in einem EF mit der Struktur „execute" abgelegt.

Aufgrund der hohen Anforderungen betreffend eine eindeutige Identifizierung von Mikrocontroller, Betriebssystem und intern Software-Schnittstellen sowie des System-Managements ist es üblich, den Download von Programmen nur online zu einem Hintergrundsystem durchzuführen. Die dort befindlichen Datenbanken enthalten entweder alle notwendigen Informationen in Abhängigkeit von einer eindeutigen Chipnummer oder erhalten diese Informationen online in einer Ende-zu-Ende-Verbindung direkt von der Chipkarte. In Abhängigkeit davon wird dann ein vorhandenes Programm mit der gewünschten Funktionalität ausgewählt und mit den geforderten Sicherheitsmechanismen zur Chipkarte übertragen. Die geheimen Schlüssel dazu werden im Hintergrundsystem üblicherweise ausschließlich innerhalb eines Sicherheitsmoduls verwaltet und benutzt. Der übliche Ablauf dazu ist im nachfolgenden Bild nochmals aufgezeigt.

Die oben beschriebene Art und Weise der Einbringung von nativem Programmcode in Chipkarten besitzt einige für die Praxis attraktive Vorteile. Das Verfahren ist unkompliziert, robust und mit geringem Aufwand an Programmcode in einem Chipkarten-Betriebssystem realisierbar. Der ausgeführte Programmcode muß nicht interpretiert, sondern kann direkt vom Prozessor abgearbeitet werden. Daraus resultiert eine hohe Ausführungsgeschwindigkeit sowie die Möglichkeit, auch komplexe Algorithmen (z.B.: DES, IDEA o.ä.) mit diesem Verfahren nachzuladen.

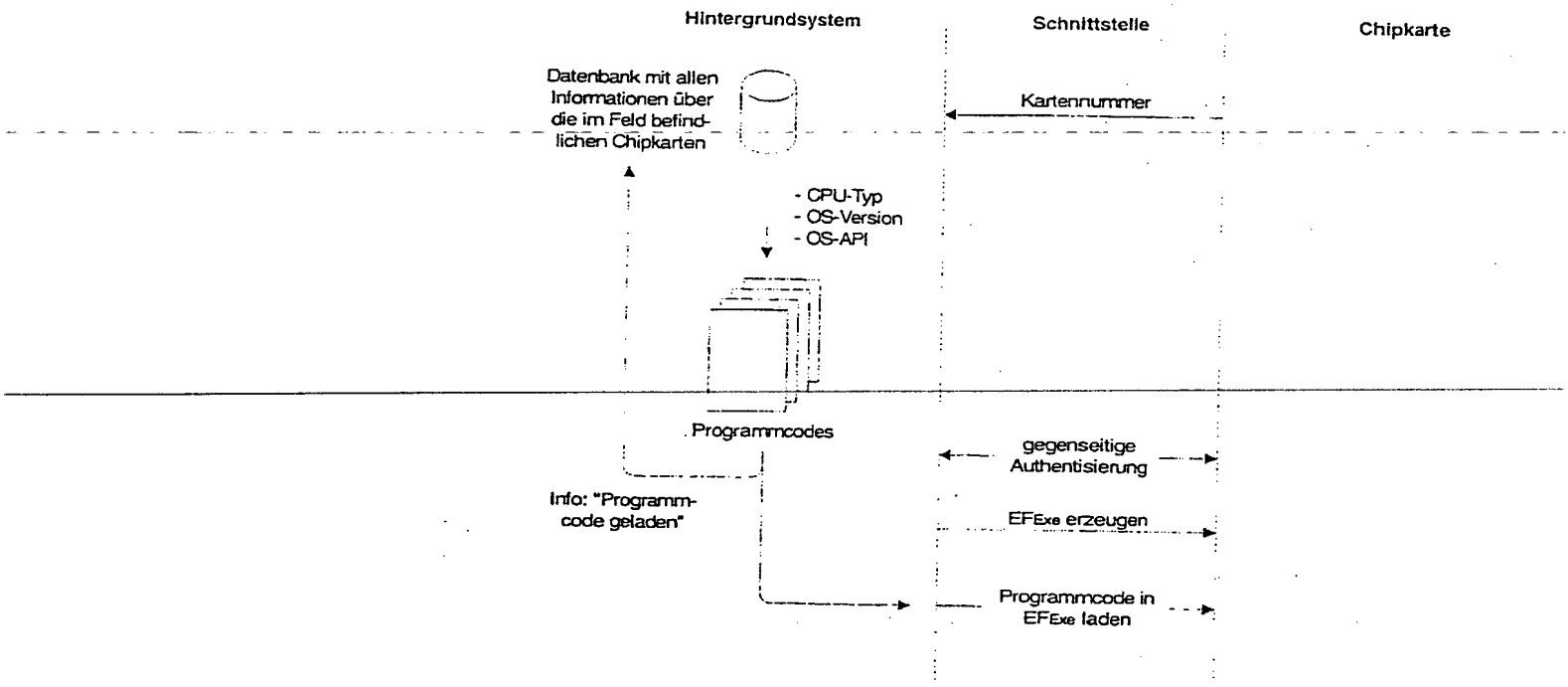

Bild 5.33 Ablauf beim online-Nachladen von nativen Programmcodes in EFs einer Chipkarte von einem Hintergrundsystem aus. Es kann bei diesem Schema abhängig vom Chipkarten-Betriebssystem und der Mikrocontroller-Hardware unterschiedlicher Programmcode zur Chipkarte übertragen werden. Der dargestellte Ablauf könnte beispielsweise sehr gut in einer GSM-Anwendung realisiert werden.

Interpreter-basierte Systeme können dies durch die geringe Abarbeitungsgeschwindigkeit des Programmcodes auch auf absehbare Zeit nicht leisten. So lange keine hardwarebasierte Speicherverwaltung (MMU) die wahlfreien Speicherzugriffe einschränkt, kann dieses Verfahren hervorragend für die Behebung von Fehlern in der Chipkarten-Software nach Kartenausgabe genutzt werden. Dies ist für den Fall des Falles eine Hintertür, die einzig und allein durch diese Art der Softwarenachladung erreichbar ist, da beispielsweise Technologien wie Java auf Chipkarten eine strikte und unbedingte Speicherseparierung verwirklichen. Ist eine MMU vorhanden, dann könnte ggf. immer noch über einen Administrator-Modus die Speicherüberwachung vorübergehend deaktiviert werden.

Dies führt nun auch schon zu den Nachteilen. Das Nachladen von ausführbaren nativen Programmen bedingt großes Wissen um die eingesetzte Hardware bzw. das Betriebssystem der Chipkarte. Unter Umständen muß für jede dieser Varianten ein eigenes Programm gleicher Funktionalität vorgehalten werden. Die zweite große Schattenseite dieser Methode ist, daß sicherheitstechnisch die Notwendigkeit besteht, daß nur der Kartenherausgeber den Programmcode erstellen oder erstellen lassen kann. Es muß strikt verboten sein, fremde und unbekannte Programme in die Chipkarte zu laden, da

diese ab dem Start das Kommando über den Mikrocontroller ohne weitere Reglementierungsmöglichkeiten erhalten. Sie könnten dann beispielsweise die geheimen Schlüssel der anderen auf der Karte befindlichen Anwendungen auslesen und über die I/O-Schnittstelle zum Terminal senden.

Einen schwachen Schutz vor Angriffen über diesen Mechanismus stellen Evaluierungen des Programmcodes durch den Kartenherausgeber dar. Besser jedoch ist in diesem Fall eine hardwarebasierte Speicherverwaltung, die dem nachgeladenen Programm nur einen bestimmten Bereich im RAM und EEPROM zur Verfügung stellt und bei dem Versuch der Überschreitung das gestartete Programm sofort beendet.[1] Damit können die einzelnen auf der Chipkarte befindlichen Anwendungen vollständig voneinander abgeschottet werden. Zur Zeit behilft man sich jedoch noch mangels entsprechender MMUs mit der Prüfung der nachzuladenden Programme.

5.10.2 Java Card

Im Jahr 1996 wurde von Europay ein Papier über OTA (*open terminal architecture*) vorgestellt, in welchem ein Forth Interpreter für Terminals beschrieben und in weiten Teilen spezifiziert war. Der Zweck war, eine einheitliche Softwarearchitektur für Terminals zu schaffen, um die Grundlage für eine hardwareunabhängige Terminalprogrammierung zu schaffen. Dann müßte man eine bestimmte Anwendung (z.B. Bezahlen mit Kreditkarte) nur noch ein einziges Mal programmieren, und diese Software würde auf allen Terminals der verschiedenen Hersteller unverändert laufen. Dieses Modell wurde aber bisher nie vollständig realisiert, es sorgte allerdings in der Chipkartenwelt für ausführliche Diskussionen.

Als dann im Herbst 1996 bekannt wurde, daß die Firma Schlumberger eine Chipkarte entwickelt, welche Programme abarbeiten kann, die in der Programmiersprache Java erstellt sind, hielt sich das Erstaunen in Grenzen. Das Prinzip eines Interpreters auf speicherplatzarmen Mikrocontrollern war durch die Open Terminal Architecture (OTA) schon reichlich bekannt. Die veröffentlichte Spezifikation Java Card 1.0 sah zur Einbindung von Java in das ISO/IEC 7816-4 Betriebssystem ein dazugehöriges API (*application programming interface*) vor, so daß von Java aus auf das bei Chipkarten übliche Dateisystem mit MF, DFs und EFs zugegriffen werden konnte.

Nach kurzzeitiger Verwunderung vieler Hersteller von Chipkarten-Betriebssystemen, warum eine Sprache wie Java, die einen üblichen Speicherbedarf weit jenseits von einem Megabyte hat, auf Chipkarten verwendet werden soll, kam es aber bereits im Frühjahr 1997 zu einem ersten Treffen von nahezu allen großen Chipkartenherstellern und der Firma Sun, die Java entwickelt und bekanntgemacht hat.

Dies war die erste Konferenz des inzwischen sogenannten Java Card Forums (JCF), welches das international tätige Standardisierungsgremium für Java auf Chipkarten ist. Die Aufgabe der technischen Gruppe des Java Card Forums ist es, eine Untermenge von Java für Chipkarten festzulegen, den Rahmen für den Java Interpreter (d.h. die *Java Virtual Machine – JVM*) zu spezifizieren und sowohl ein allgemeines, als auch an-

[1] siehe auch Abschnitt 3.4.3 Zusatzhardware

Information Disclosure Statement with Regard to the Non-English Language Document

In Ranke, Wolfgang et. al, „Handbuch der Chipkarten. Aufbau - Funktionsweise - Einsatz von Smart Cards", Carl Hanser Verlag, Munich 1999), pages 191 - 193, various authentication methods between chip card and terminal are described, and on pages 252 - 261, the above document relates to chip-card operating systems with a reloadable program code. Various advantages are set forth which result from the reloadability of program codes in chip cards, e.g. the possibility of removing program errors in fully personalized cards (page 252, last line, up to page 253, first line), and the possibility of an application provider introducing an encryption algorithm only known to themselves into the chip card (page 252, second but last paragraph). Two ways of reloading are indicated, specifically reloading in the form of native code or in the form of an executable program code to be interpreted (picture 5.27). The problems associated with reloading native codes, i.e. the lacking possibility of monitoring a downloaded executable code (page 254, third but last paragraph), are said to be solvable by mutual authentication prior to the actual loading of the program code (page 254, third but last line) and by providing an MMU (page 255, third paragraph). Complex algorithms such as DES, IDEA are given as examples of reloadable native codes (page 259, last two lines).

Unlike the subject-matter of the newly submitted independent claims, the above document does not disclose a volatile memory for storing part of an algorithm code which is received via a data interface. In addition, like US 4,777,355, the above document does not address the issue of the chip card's energy supply.

DOCKET NO: S&ZIO020101
SERIAL NO: 10/620,108
APPLICANT: Janke

LERNER AND GREENBERG P.A.
P.O. BOX 2480
HOLLYWOOD, FLORIDA 33022
TEL. (954) 925-1100